THE DOOM OF DEVICE DRIVERS: Your Android Device (Most Likely) has N-Day Kernel Vulnerabilities

Lukas Maar
Graz University of Technology

Florian Draschbacher
Graz University of Technology and A-SIT Austria

Lorenz Schumm
Graz University of Technology

Ernesto Martínez García
Graz University of Technology

Stefan Mangard
Graz University of Technology

Abstract

Android's security landscape is constantly evolving to counter increasingly sophisticated attacks, with the kernel as a prime focus. Past device compromises required complex exploit chains pivoting to privileged contexts before targeting the kernel. Recently, however, the trend has been to exploit kernel GPU drivers accessible to untrusted apps to bypass privileged pivoting. While significant efforts have been made to secure GPU drivers, the broader risks of untrusted apps compromising Android devices remain underexplored at a large scale.

In this paper, we perform the first comprehensive analysis of kernel drivers accessible to untrusted apps on a representative set of 131 Android devices. Using our mostly automated approach to recover access control policies from device firmwares, we identify a significant attack surface beyond GPUs, comprising 11 drivers. From public information about these drivers, such as git repositories, we reconstruct 50 known vulnerabilities, including highly critical issues that allow exploit primitives such as use-after-free and out-of-bounds writes. Our subsequent vulnerability patch inclusion analysis reveals that many of these vulnerabilities remain unpatched, acting as n-days at the time of analysis[1] or for extended periods: More than 59 % of the analyzed devices can be exploited by highly critical n-day vulnerabilities.

We uncover novel insights into the disparity in patch timelines and vendor practices. Our findings show that malicious actors can exploit n-day vulnerabilities accessible to untrusted apps, bypassing the need for complex zero-day vulnerabilities. We conclude that urgent action must be taken to improve overall Android security.

1 Introduction

In today's interconnected world, mobile phones are essential to daily life, with Android powering billions of devices. Ensuring their security is critical, as compromises can allow surveillance, expose sensitive data, or enable identity theft. With increasingly sophisticated attacks, timely vulnerability identification and mitigation are becoming more critical.

Past full Android device compromises required complex exploit chains, as Android tightly restricts the kernel attack surface exposed to most apps. Consequently, these exploit chains often pivoted through elevated processes before targeting the kernel. For example, a 2022 attack detailed by Google Project Zero [27] began with remote code execution in Chrome. Like most apps, Chrome runs in an untrusted security context with only a limited kernel attack surface. The attack exploited a system service vulnerability to gain system privileges, increasing the reachable kernel attack surface. Finally, with system privileges, it targeted a kernel sound driver vulnerability for an arbitrary read/write, enabling device compromise. Such full chains are commonly used to secretly install spyware—like Pegasus [22] or the newly discovered NoviSpy [23]—which is eventually used for surveillance [76].

However, recent kernel attacks bypass the need for complex chains. Experts [8, 14, 16–18, 56–58, 73, 75, 76, 84, 87] highlighted that GPU drivers are directly accessible from untrusted contexts. Hence, full-chain exploits are now targeting the GPU directly from untrusted apps, eliminating the need for privilege pivoting. This strategy shift has made GPU drivers prime targets. With four GPU suppliers covering the entire Android market—ARM Mali, Qualcomm Adreno, Samsung Xclipse, and Imagination Technologies PowerVR—a single vulnerability in any of the drivers can impact a wide range of devices. Additionally, the inherent complexity of GPU drivers made them particularly susceptible to vulnerabilities.

Google's 2023 annual review [76] underscored this, attributing 4 of 5 device compromises to GPU driver vulnerabilities, with only 1 involving the core Linux kernel. In response, Google has prioritized GPU security [87], collaborating with the Android Red Team and ARM to enhance GPU vulnerability detection, mitigation, and hardening. This raises a critical question: *Are GPUs the sole attractive target for malicious actors, or do other kernel components pose similar or even greater risks that have yet to be addressed comprehensively?*

In this paper, we comprehensively analyze the kernel attack

[1] December 2024: time of analysis.

surface accessible to untrusted apps and show that multiple kernel drivers remain vulnerable to n-day exploits, i.e., exploiting vulnerabilities that remain unpatched despite fixes being known. At the time of our analysis, 59.1 % of recent Android devices in our representative set can be exploited by highly critical n-day flaws, with 61.4 % affected by vulnerabilities of any severity. Highly critical flaws include Use-After-Free (UAF) [49, 60, 86] and Out-Of-Bounds (OOB) writes [9, 84], while moderate ones include Uninitialized Variables (UV) [11, 38, 47] and Information Disclosures (ID) [43, 44, 48]. Our findings show that these n-day driver vulnerabilities are even more attractive targets than GPU ones, as they are similarly accessible from untrusted apps and affect multiple devices but remain unpatched for extensive time. Malicious actors can, therefore, exploit these n-day vulnerabilities without needing to find zero-day vulnerabilities. By highlighting this gap, we envision improving the security maintenance of device drivers and ultimately enhancing Android security.

To achieve this, we perform three analyses: First, we analyze the kernel attack surface of drivers accessible to untrusted apps. Second, we reconstruct vulnerabilities in these drivers using public information. Third, we evaluate the vulnerability patch inclusion, which indicates the prevalence of n-days.

For the kernel attack surface analysis, we present a mostly automated approach for extracting access control data from device firmwares. This method recovers Linux's user-group-based access control and SELinux's policies, which form the fortified environment isolating security domains [2, 54]. Using this, our approach identifies kernel drivers accessible from untrusted security contexts, where most apps run. We analyze 493 firmwares from the most recent 131 devices of 7 OEM vendors (Samsung, Xiaomi, Asus, Realme, OnePlus, Oppo, and Vivo). These OEMs represent more than 75 % of Android's market share. Our results show that, apart from GPUs, 11 drivers are accessible from untrusted contexts, including components like the DSP, JPEG decoder, and AI coprocessor.

For driver vulnerability reconstruction, we collect publicly available git repositories and bug reports for 7 of the 11 drivers from two chipset ODM vendors (Qualcomm and MediaTek), covering low- to high-end devices. We then reconstruct 50 vulnerabilities from the last 4 years by searching for security-relevant keywords and manually identifying security patches.

For the patch inclusion, we semi-automatically detect the absence of patches for 21 of our 50 identified vulnerabilities. Analyzing 493 firmware images across multiple OEM vendors, we find that many vulnerabilities remain unpatched for extended periods, some exceeding one year, while others are still unpatched at analysis time. Notably, 61.4 % of devices are affected by at least one known vulnerability, with 59.1 % exposed to highly critical issues. We support n-day exploitability by triggering 5 such n-day vulnerabilities from the DSP driver on a real device, highlighting cross-OEM susceptibility.

We present 5 key findings: *First*, if a driver contains one n-day vulnerability, it is highly likely to contain more. For example, 71.4 % of our Xiaomi devices have at least one, while 49 % contain three or more. *Second*, OEMs primarily address vulnerabilities by releasing new devices rather than issuing updates for existing ones. *Third*, patch delays vary widely across OEMs, ODMs, and vulnerability types, with OOB experiencing the fastest patch inclusion, followed by UAF and ID. When comparing ODMs, MediaTek devices are more than 2 times slower to receive patches than Qualcomm devices. *Fourth*, n-day proof-of-concept exploits targeting these drivers are versatile and can be reused across OEMs. *Fifth*, the presence of n-day vulnerabilities in drivers accessible to untrusted apps enables exploitable pathways, reducing the need for time-consuming zero-day development.

In conclusion, our findings highlight the urgent need for stronger defensive measures in Android security, especially as concurrent research [23, 24, 33] reveals that malicious actors actively exploit accessible drivers in the wild.

Contributions. The main contributions of this work are:

(1) Comprehensive Kernel Attack Surface Analysis: We present the first comprehensive analysis of kernel drivers accessible to untrusted apps, identifying a broader attack surface beyond GPU drivers, comprising 11 drivers.

(2) Reconstruction of Kernel Vulnerabilities: We reconstruct 50 vulnerabilities, including high-critical issues like use-after-free and out-of-bounds writes, showing the prevalence of exploitable vulnerabilities in drivers.

(3) N-Day Patch Inclusion Analysis: We conduct a semi-automated analysis that reveals significant patch delays, with 59.1 % of devices vulnerable to high-critical n-day exploits, demonstrating persistent security gaps.

(4) Insights into Vulnerability Trends: We uncover that n-day kernel driver vulnerabilities are more attractive to malicious actors than GPU vulnerabilities and highlight disparities in patch timelines and vendor practices.

Outline. Section 2 provides background. Section 3 shows the high-level overview. Section 4 presents the kernel attack surface analysis to untrusted apps. Section 5 reconstructs vulnerabilities. Section 6 detects patches, showing multiple vulnerabilities act as n-days. Section 7 discusses security implications and related work. Section 8 concludes our work.

2 Background

This section covers the kernel exploitation terminology, Generic Kernel Image (GKI) project, Android's access control, and full-chain exploits targeting Android devices.

Kernel Exploit Terminology. We refer to exploit-specific definitions from prior work [6, 48]. A *zero-day* exploits a *zero-day vulnerability* before it is publicly disclosed or patched, while an *n-day* targets an *n-day vulnerability*, a known security issue with existing patches or mitigations that may not yet be applied. A *full exploit chain* consists of multiple stages that typically exploit a messenger [25, 61] or browser [27] and progress to the kernel with intermediate stages, ultimately

compromising the device. *Exploit primitives* are basic capabilities obtained through vulnerability exploitation (e.g., out-of-bound writes), and *exploit techniques* convert primitives into more impactful outcomes (e.g., an arbitrary read/write).

Generic Kernel Images and Kernel Drivers. The Android OS is based on the Linux kernel, which faced challenges adapting to different devices. Before the GKI project [4], OEM vendors maintained product kernels for each device model, derived from the upstream Android Linux kernel and heavily modified. The wide range of devices resulted in a large number of product kernels and kernel fragmentation, which had negative security consequences. These include significant delays in rolling out security-critical updates—also highlighted by prior work [10, 35, 64, 89, 95]—or difficulty in merging upstream changes. To counter this trend, Google initiated the GKI project [4]. With GKI 1.0, introduced in Android version 11, devices running 5.4 kernels must pass GKI tests., i.e., from the compatibility test suite. With GKI 2.0, devices running 5.10 or later kernels must ship with the GKI kernel maintained and built by Google. GKI 2.0 has security benefits, as these kernels are updated with long-term stable changes and critical bug fixes, resolving the kernel fragmentation issue. To compensate for device customization, OEMs now rely on introducing customization via kernel modules.

SELinux and Android's Access Permissions. Android combines Linux's user-group-based access controls with Security-Enhanced Linux (SELinux)'s mandatory policies, creating a fortified environment where different security domains are isolated [2, 54]. This reduces the risk of malicious interference and enhances the overall system security.

Linux offers Discretionary Access Control (DAC) for managing file system permissions so that users cannot alter or access other users' resources, which Android uses to isolate applications from each other. Each app runs under its own Linux user, and files created by one app cannot be accessed by other apps unless explicitly granted permission to share. For more fine-grained access control, Android establishes Mandatory Access Control (MAC) on processes with SELinux. SELinux offers this by integrating into the Linux Security Module (LSM) framework and using syscall hooks and policies to enforce access control decisions. The system follows a default-denial principle, allowing only explicitly permitted actions. It operates in permissive or enforcing mode, where, per default, Android runs in enforcing mode.

SELinux policies define rules for allowing actions by a particular object on a specific subject. The subject is commonly a set of processes that run in the same security domain, also called a security context. On Android, the untrusted security context (i.e., `untrusted_app`) is the domain assigned to third-party applications installed from the Google Play Store or other sources. It ensures that apps are restricted from performing unauthorized actions or accessing sensitive system resources, e.g., most of the kernel and its drivers. This context prevents apps from directly interacting with other applications, enforcing strict boundaries unless explicitly permitted by mechanisms like Inter-Process Communication (IPC), e.g., through Android's binder IPC. In addition to `untrusted_app`, Android defines other SELinux contexts for different types of apps or system components, such as `system_app` for privileged apps. The SELinux policies for each context ensure that processes operate within predefined boundaries, minimizing security risks and maintaining system integrity.

3 High-Level Overview

This section provides an overview of our work, first highlighting how past full-chain exploits have primarily targeted Android devices. While prior attacks typically exploited well-studied vulnerabilities in GPU drivers or the kernel alongside privileged process exploitation, we analyze alternative, under-explored attack surfaces for large-scale root compromise.

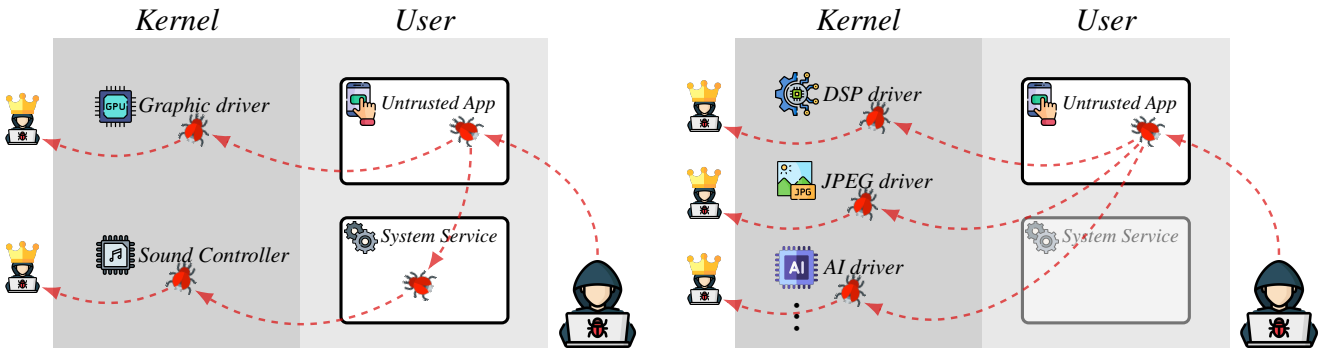
Prior Exploitation Chains. Full-chain Android exploits (see Figure 1a) typically achieve code execution in an untrusted security context by exploiting vulnerabilities in an application [73, 75, 76], such as browsers [27] or messengers [25, 61]. With code execution, these exploits have typically followed one of two pathways to root, compromising the device: First, the attack targets a vulnerability in a higher-privileged process, allowing it to elevate from the untrusted to the higher-privileged security context, e.g., the system context. This escalation significantly increases the kernel attack surface. With higher privileges, the attack then exploits one or more kernel vulnerabilities that are accessible from this context [27, 36, 46, 72], such as the `io-uring` subsystem [46] or the higher-privileged sound device drivers [27]. Second, the attack targets one or more vulnerabilities that are accessible from the reduced kernel attack surface within the untrusted context. In this case, malicious actors mainly focus on vulnerabilities in the GPU driver. In fact, according to Google's annual report in 2023 [76], GPU drivers were targeted by 4 out of 5 full-chain exploits, with one taking the first pathway.

From an attacker's perspective, both approaches face a similar problem: Security researchers are aware of them and have made significant advances in suitable detection and mitigation. For instance, collaborative efforts by Google, the Android Red Team, and ARM have substantially enhanced the security of Android GPU drivers [87]. These improvements have largely concentrated on GPU driver vulnerabilities, leaving other kernel components less explored and vulnerable.

Presented Exploitation Chains. In contrast to the past focus on GPU drivers, we identify and analyze alternative kernel components as equally—if not more—critical exploit targets (see Figure 1b). We show that these components meet the following generalized criteria for exploitation:

C1: Accessibility. The target kernel component is directly accessible from untrusted security contexts.

C2: Broad Impact. A vulnerability in the target component can affect a wide range of Android devices.



(a) **Prior Exploitation Chains:** Attack directly the graphics driver from an untrusted or the kernel from a system security context.

(b) **Presented Exploitation Chains:** Attack directly kernel drivers which are accessible from an untrusted security context.

Figure 1: Exploitation chains of attacking Android devices to get full root.

C3: Susceptibility. The target is highly susceptible to including unintentionally exploitable vulnerabilities.

In Section 4, we present a mostly automatic approach to identify other components accessible from untrusted contexts. Our findings reveal that beyond GPUs, 11 device drivers, such as those for the Digital Signal Processor (DSP), JPEG decoding accelerator, and Artificial Intelligence (AI) coprocessor, meet **C1**. In Section 5, we perform a semi-automated analysis to find n-day vulnerabilities by inspecting publicly available git repositories and bug reports. We identify 50 vulnerabilities within 7 device drivers that affect a wide range of devices, satisfying **C2**. In Section 6, we reveal that many identified n-day vulnerabilities remain unpatched for an extensive amount of time or unpatched till December 2024, i.e., the date of the analysis. This leaves 61.4 % of devices vulnerable, with 59.1 % exposed to highly critical vulnerabilities. The lack of n-day patches eliminates the need for the time-consuming discovery of complex zero-days, satisfying **C3**.

Threat Model. In our threat model, we assume a malicious actor who has already achieved code execution in an untrusted security context, e.g., by exploiting an application like Chrome. The malicious actor's goal is to compromise the device's kernel and take full control of the device with minimal effort and resources. Given the time and resource intensity of discovering zero-day vulnerabilities, the malicious actor aims for alternative pathways, including the exploitation of n-day vulnerabilities. This aligns with the expectations of real-world Android exploitation [27, 53, 68, 70, 72].

Collection and Extraction of Firmwares. We automatically collect firmwares not protected by captchas and manually collect those that are protected by captchas. We implement a web crawler based on Python Selenium to download firmwares from different points in time where possible. We consider 7 OEM vendors, accounting for more than 75 % of the Android market [5]. These OEMs comprise the top 5 (Samsung, Xiaomi, Vivo, Oppo, Realme) as well as 2 well-recognized (OnePlus, Asus), and use the chipset of 3 ODM vendors (Qualcomm, MediaTek, Samsung). We consider devices of OEMs released between October 2022 and December 2024 (completion of the analysis). Our focus lies on recent devices as these are more likely to receive security updates [1, 48, 95]. Overall, we collect and extract 493 firmwares for 131 Android devices which is a representative sample of the 488 devices produced in this time span. For Samsung and Xiaomi, our collection includes version lineages, i.e., multiple different firmware versions for the same device.

4 Attack Surface Analysis of Android Kernels

This section conducts a large-scale analysis to identify the kernel attack surface accessible for the untrusted security context. This involves generalizing Android's approach to minimizing the kernel attack surface and detailing how this information can be extracted from device firmwares (see Section 4.1). Using this approach, we analyze the kernel attack surface of 493 firmwares (see Section 4.2). We show that this surface includes multiple kernel devices, satisfying criteria **C1**. To validate these findings, we perform dynamic access tests (see Section 4.3) by implementing an application in the untrusted context to confirm access to the previously identified drivers.

4.1 Determining the Minimum Kernel Attack Surface

To determine the kernel attack surface accessible to untrusted contexts, we address two key questions: What kernel components are potential attack targets, and which are accessible to unprivileged apps? We focus on device drivers, the most vulnerable part of the kernel [7, 52]. Hence, to determine the attack surface of kernel drivers, we need the drivers themselves and each access permission.

Prior to GKI 2.0, these kernel drivers were typically included in the kernel binary. However, starting GKI 2.0, OEMs

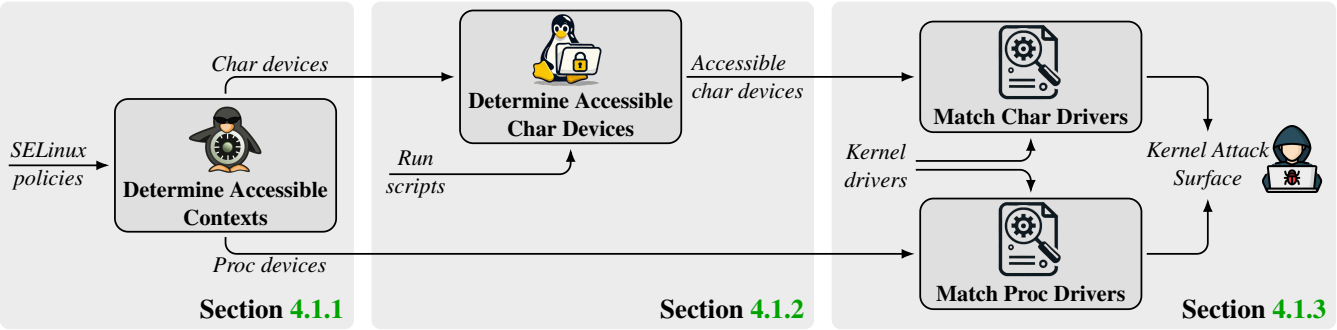


Figure 2: High-level workflow for determining the kernel attack surface from SELinux policies, Linux permissions and drivers.

were forced to use the generic Android kernel image, implying that ODM-specific drivers are no longer included in the binary. Instead, they are loaded as kernel modules typically during boot. The storage locations of these drivers embedded within the device's firmware vary and depend on the OEM and model. After obtaining the drivers, we determine which are accessible to untrusted security contexts. To achieve this, we extract two access control data from the firmware: SELinux policies and user-group-based Linux permission settings, both defining the access to these drivers.

Figure 2 illustrates our high-level workflow for determining the kernel attack surface by analyzing the SELinux policies (in Section 4.1.1) and the Linux permission settings (in Section 4.1.2) to find the matching kernel drivers accessible by unprivileged security contexts (in Section 4.1.3).

4.1.1 Analyzing SELinux Policies

To reconstruct SELinux access control policies, two configuration files are critical [3]: The precompiled policies (i.e., `precompiled_sepolicy`), which configure allowed access of SELinux contexts to specific domains; and the domain mappings (i.e., `vendor_file_contexts`), which assign file paths to domains. Together, they allow the identifying actions of a SELinux context to be performed on a file at a given path. Depending on the device, the configurations are stored in three possible locations for `precompiled_sepolicy` and two for `vendor_file_contexts`, e.g., `/etc/selinux` in partition `odm` or `vendor`. Our approach extracts these policy-related files for subsequent analysis. We then use the official SELinux policy query tool, `sesearch`, to obtain access control rules for character devices from untrusted contexts.

Character Devices. Android's hardware resources are typically managed by kernel drivers that expose higher-level functionality to user space via character device files. These character devices usually mount virtual files in the `/dev` directory. User-space apps can interact with them via syscalls like `open` and `ioctl`. To find character devices accessible in the unprivileged `untrusted_app` context, we execute queries against `precompiled_sepolicy`.

```
1 allow appdomain vendor_qdsp_device:chr_file { ioctl read };
2 allow domain zero_device:chr_file { append getattr ioctl lock
      map open read watch write };
3 allow untrusted_app gpu_device:chr_file { append getattr
     ioctl lock map open read watch write };
4 allow untrusted_app sound_device:chr_file { append getattr
     ioctl lock map open read watch watch_reads write };
5 allow untrusted_app_all untrusted_app_all_devpts:chr_file {
     getattr ioctl open read write };
```

Listing 1: SELinux access control for `untrusted_app` on Xiaomi Redmi Note 14 Pro+.

```
1 /dev/kgsl              u:object_r:gpu_device:s0
2 /dev/adsprpc-smd       u:object_r:vendor_qdsp_device:s0
3 /dev/xlog              u:object_r:sound_device:s0
```

Listing 2: Mounting points for domains accessible to `untrusted_app` on Xiaomi Redmi Note 14 Pro+.

For instance: `sesearch -allow -s untrusted_app -c chr_file -p ioctl precompiled_sepolicy` finds all character devices accessible via the `ioctl` syscall. Listing 1 illustrates a simplified SELinux policy output by `sesearch` on the Xiaomi Redmi Note 14 Pro+. It shows domain-specific permissions, which control resource access for processes running with the `untrusted_app` context. The `appdomain` (a context despite its name) covers most Android apps (including `untrusted_app`) and, in this example, can access files in the `vendor_qdsp_device` domain. The broader `domain` (a confusingly named context that comprises all processes on the device) is granted permission on the `zero_device` domain. The `untrusted_app` context, a subset of `appdomain`, has stricter controls but full access to `gpu_device`. The more restrictive `untrusted_app_all` context allows access to `untrusted_app_all_devpts` on the specific firmware.

SELinux policy only allows us to learn about access to domains, e.g., `vendor_qdsp_device`. To resolve these domains to mounting points of a character device within the `/dev` directory, we use the `vendor_file_contexts`. Listing 2 illustrates the content of `vendor_file_contexts` relevant to recover the device's mounting point. For instance,

```
1 allow appdomain appdomain:binder { call transfer };
2 allow appdomain appdomain:fd use;
```

Listing 3: Transfer from `untrusted_app` to `platform_app`.

the mounting point for the `vendor_qdsp_device` domain is `/dev/adsprpc` on Xiaomi Redmi Note 14 Pro+.

ProcFS Files. The Process File System (ProcFS) provides an alternative mechanism for interacting with kernel device drivers [29]. Kernel drivers can expose user-space interfaces by creating virtual files using the `proc_create` kernel function, which takes the file name and access permissions as arguments. Accessing these files through syscalls prompts its kernel driver functions, enabling communication between the user and kernel. Similar to character devices, we use `sesearch` to find the access permissions for ProcFS files. However, we do not need `vendor_file_contexts`, as their domain names already contain the path, e.g., `allow appdomain proc_ged:file {...}` refers to the path `/proc/ged`.

Pivoting Contexts. As observed in Listing 1, permissions for character devices vary. For instance, while `zero_device` permits mapping, `vendor_qdsp_device` does not. SELinux policies may allow certain operations (e.g., `ioctl`) on a file but restrict others, such as opening (e.g., `vendor_qdsp_device`). This limitation can be bypassed by legally pivoting to contexts with open permissions for the file [21, 33], avoiding the need for a vulnerability. To identify pivoting contexts, we query contexts that allow the target device to be opened and locate transitions from `untrusted_app` to those contexts. This allows transitioning to a context that can share device references with `untrusted_app`. For example, we identified `platform_app` and `vendor_dspservice` as pivoting contexts that enable interaction with `vendor_qdsp_device`.

Listing 3 illustrates these transfers. Line 1 permits `appdomain` (including `untrusted_app`) to use Android's Binder IPC for calls and data transfer between `appdomains`, e.g., `platform_app`. Line 2 permits shared file descriptors, enabling `untrusted_app` to access resources used by `platform_app`. Thus, Binder IPC and shared file descriptors allow `untrusted_app` to open access `vendor_qdsp_device`.

4.1.2 Analyzing Linux Permission Settings

Linux executes Run Control (RC) scripts during startup to set up services and configs, including permissions. These scripts use commands like `chmod` and `chown` to define user-group-based permissions to read, write, and execute for files and devices. For character devices, the permissions are typically set based on predefined policies in scripts or config files.

We implement an approach that extracts all found RC scripts (in `/etc/` or `/etc/init`) to examine these permission settings. We mark a character device as accessible if the RC permissions permit it to unprivileged `others` users, and a SELinux policy rule allows access to unprivileged contexts. There are cases where only one is true. For instance, SELinux's access control allows the `ioctl` syscall on `/dev/sdsprpc-smd` while its permissions is `0660 system:system`, indicating that only the system user/group has read/write access to this device but no unprivileged `others` user. Another example is `/dev/elliptic`, which has `0644 system:system` permissions, but no SELinux access control rule allows access to it from the `untrusted_app` context.

4.1.3 Matching Kernel Drivers

Since GKI 2.0, OEMs are forced to use the Google-maintained GKI kernel and are required to move kernel drivers to external modules. There are 2 possible locations of kernel drivers, either in the `vendor_dlkm` partition or compressed inside a ramdisk stored within the `vendor_boot` partition. Android uses a ramdisk to initialize the system before mounting the main file systems. A ramdisk is a temporary file system loaded into RAM during the boot process. The driver extraction varies depending on the storage: If drivers are located within `vendor_dlkm`, we mount the partition and copy the drivers for further analysis. Extraction from the `ramdisk` requires decompressing the disk image, then extracting the drivers from the ASCII cpio archive using a tool like `binwalk`. `binwalk` extracts a file system from the cpio archive, storing the drivers in `/vendor` or `/vendor_dlkm`.

With the reconstructed driver modules and the list of accessible device nodes in the file system, we match each device node—either in `/dev` or `/proc`—to its driver. Our approach depends on the device node's mounting point. For `/dev`, we automatically match the file name (e.g., `adsprpc-smd`) with all strings contained in kernel modules. We then manually verify the mapping by comparing it with the driver's source code. For `/proc`, we automatically scan driver modules for the file name (e.g., `jpeg_driver`) and the `proc_create_file` symbol. We then manually inspect the identified kernel module and its source code. This verifies the mapping and confirms that the `mode` argument passed to `proc_create_file` renders the device node accessible to untrusted contexts.

By combining these methods, we establish a mapping of each mounting entry to its corresponding kernel driver. To ensure that these drivers are loaded at startup, we verify that the matched module appears in the `modules.load` file, which lists all kernel modules to be loaded automatically.

4.2 Large-Scale Analysis

This large-scale analysis is a fully automated process to determine all kernel drivers that are accessible to untrusted contexts. The manual preprocessing phase, outlined in Section 4.1, serves as an initial phase for this automated analysis, where we map each driver's entry point to its corresponding kernel module. The automated process analyzes SELinux

Table 1: Accessible kernel attack surface from the untrusted security contexts, showing the percentage of devices per OEM vendor permitting access to the corresponding kernel driver.

Category	Device Driver's Entry	Module	Accessibility per OEM Vendors						
			Samsung	Xiaomi	Asus	Realme	OnePlus	Oppo	Vivo
AI	/dev/apuext	apusys.ko	2	12		12		29	40
DSP	/dev/adsprpc-smd	frpc-adsprpc.ko	48	52	83	56	71	43	40
DSP	/dev/fastrpc-[acs]dsp	frpc-adsprpc.ko		10					
NPU	/dev/vertex	npu.ko	38						
AI	/dev/apusys	apusys.ko		4					
Audio	/dev/xlog	xlogchar.ko		60					
GPU Extention	/proc/ged	ged.ko	14	38	17	38	29	57	60
Monitor	/proc/perfmgr	mtk_perf_ioctl.ko	7	38	17	38	29	57	60
JPEG	/proc/mtk_jpeg	jpeg-driver.ko	2	25	17		14	29	40
Memory	/proc/secmem	trusted_mem.ko	12	25					
Monitor	/proc/mi_log	mi_log.ko		21					
Camera	/proc/camera	camera.ko	10						

policies, Linux permission settings, and kernel drivers across 493 firmware versions of 131 Android devices from 7 OEMs. This analysis determines drivers that untrusted contexts can access. Table 1 presents the results, highlighting the extent of the kernel attack surface across OEMs. These findings reveal that multiple drivers are accessible and that most remain accessible across OEMs, satisfying **C1**.

Table 1 categorizes the accessible kernel drivers based on their intended functionality (derived from `modinfo`) and lists their entry points and module names. For each OEM and driver module, the table indicates the percentage of devices permitting access or, if access is absent, leaves the cell blank. The most contributing reason for inaccessibility is the lack of hardware support for the corresponding software driver module. For instance, the `npu.ko` driver is found exclusively on Samsung-ODM devices, present in 38 %. Another less dominant contributing factor to variability in access is hardware support combined with device-specific access permission settings. For example, while the `apusys.ko` driver is included in devices from Xiaomi, Oppo, OnePlus, and Realme, only a subset of Xiaomi devices allows access to untrusted contexts.

Device configurations show mutually exclusive driver sets based on the ODM chipset. For example, Xiaomi devices are either Qualcomm- or MediaTek-based, leading to Qualcomm drivers such as `/dev/adsprpc-smd` or `/dev/fastrpc-[acs]dsp`, or MediaTek drivers such as `ged.ko` and `mtk_perf_ioctl.ko`. This pattern extends to other OEMs. Among Xiaomi's MediaTek devices (38 % of the lineup), about 65 % include the `jpeg-driver.ko` driver. Those findings highlight the nuanced variability in kernel driver accessibility across devices, ODMs, and OEMs.

4.3 Validity of Analysis

Our analysis relies on static interpretation of kernel drivers, SELinux policies, and RC scripts, all of which contribute to driver accessibility from unprivileged contexts. To verify the validity of our statically determined results, i.e., to confirm they reflect behavior on real devices, we perform dynamic testing on a representative subset of Android devices. We pick 15 devices from 4 OEM (Samsung, Xiaomi, Vivo, Oppo) and 3 ODM vendors, reflecting the 4 most popular Android OEMs by market share [5]. To span the performance spectrum, we test 3 Samsung models (S24 Ultra, A55, A14), and the Xiaomi Redmi 12, Vivo Y36, and Oppo A58. Additionally, we validate driver accessibility on nine more Samsung devices via the Remote Test Lab. In total, our test set includes 5 Qualcomm-based, 6 MediaTek-based, and 4 Samsung ODM.

To determine device driver interfaces accessible from untrusted contexts at runtime, we implement an unprivileged Android application. It attempts to invoke a series of syscalls on each file in `/dev/` or `/proc/` whose accessibility we wish to determine. These syscalls are `open`, `close`, `read`, `write`, `ioctl`, `fgetxattr`, `mmap` and `flock`, representing SELinux's access permissions (see Listing 1). We consider a file as accessible if the syscall yields success or the resulting error code does not indicate a lack of permission. Listing the contents of `/dev/` or `/proc/` from an untrusted context, such as our test app, may be forbidden, even if access to the contained files is possible. Hence, we list directory contents as the higher-privileged shell user. We pass the obtained list of files to our unprivileged app to test the syscalls. For some subfolders of `/dev/` or `/proc/`, not even the shell user may list contents. As this only affects a small number of paths across all firmwares, we hardcoded them into the app.

For all evaluated devices, the results of our static analysis align with those obtained at runtime. Hence, we conclude that our large-scale results estimate real devices.

5 Analysis of N-Day Vulnerabilities

In this section, we present a systematic analysis of n-day vulnerabilities. These are security flaws that have been publicly disclosed and have available patches but remain unpatched on devices. While some security flaw sources—such as public

vulnerability disclosures [87] and write-ups [58]—explicitly reveal the nature of the flaw, others—such as Security Bulletins from Google or Qualcomm—provide less direct information, making the identification process more complex.

A notable observation in our analysis is that most publicly available write-ups and vulnerability disclosures focus disproportionately on GPU driver vulnerabilities [18, 55–58, 76, 87]. This focus has led to significant progress in understanding and addressing GPU-related security issues. However, it has also created a gap in public knowledge about exploiting vulnerabilities in other types of drivers, e.g., other drivers accessible from untrusted contexts. To address this imbalance and ensure a broader vulnerability exploitation coverage, our approach extends beyond GPU drivers and analyzes driver-specific vulnerabilities. We achieve this by identifying vulnerabilities using a history tree search across different drivers. Through this analysis, we successfully identify 50 vulnerabilities, detailed in Table 2. Section 6 then shows that multiple of these vulnerabilities are either n-days at the time of analysis or for extended periods. By showing that each of these n-day vulnerabilities affects multiple devices, it meets **C2**.

5.1 N-Day Vulnerability Identification

Android adheres to a strict open-source policy to ensure transparency and encourage collaboration within the security community. This includes maintaining public access to *bug reports* and releasing *kernel modifications*. While this openness empowers security researchers to identify, analyze, and address potential vulnerabilities, it also provides malicious actors with the means to identify n-day vulnerabilities.

To identify such n-day vulnerabilities, *bug reports* can potentially serve as a direct source of information. Ideally, these reports should only be publicly available after the associated vulnerabilities have been patched in all systems, including downstream versions. However, in practice, they typically become public after exceeding the disclosure deadline or following a grace period after a patch is released. Google Project Zero, for instance, follows a 90+30 disclosure deadline policy. If a patch is released within a 90-days time period, details are disclosed 30 days later. If no patch has been released, the reports are publicly disclosed after 90 days.

Bug reports can be generalized into two categories: First, vulnerability disclosures reveal official disclosure information, which provides detailed vulnerability descriptions. Examples include CVE-2022-22706/CVE-2021-39793 [20] and issue trackers highlighting CVE-2024-23384 [92] and CVE-2024-23698 [12]. Second, exploit write-ups and analyses encompass publicly available exploit details contributed by the security research community. Examples include works by Mo [55–58] and studies of zero-day and n-day vulnerabilities found in the wild, e.g., done by Google Project Zero [29]. As described above, both categories of bug reports provide direct information about vulnerabilities and, if not patched, offer the possibility of exploiting them in an n-day scenario.

```
commit 2466bcf3cea4ed9b37b7e8983e7e6b7ffd92e8fc
Author: quic_anane <quic_anane@quicinc.com>
Date:   Tue Jul 16 23:37:45 2024 +0530

    msm: ADSPRPC: Avoid Out-Of-Bounds access

    Currently, when adding duplicate sessions to an array that
    holds session information, no check is performed to avoid
    going out of bounds. Add a check to confirm that the index
    is not out of bounds.

    Change-Id: Ib7abcc5347ba49a8c787ec32e8519a11085456d9
    Signed-off-by: quic_anane

diff --git a/dsp/adsprpc.c b/dsp/adsprpc.c
index d7e2c3e..631d1b3 100644
--- a/dsp/adsprpc.c
+++ b/dsp/adsprpc.c
@@ -8172,6 +8172,12 @@ static int fastrpc_cb_probe(struct
      device *dev)
      for (j = 1; j < sharedcb_count &&
          chan->sesscount < NUM_SESSIONS; j++) {
        chan->sesscount++;
+       VERIFY(err, chan->sesscount < NUM\_SESSIONS);
+       if (err) {
+         ADSPRPC_WARN("failed to add shared session, maximum
      sessions (%d) reached \n", NUM_SESSIONS);
+         break;
+       }
        dup_sess = &chan->session[chan->sesscount];
        memcpy(dup_sess, sess,
          sizeof(struct fastrpc_session_ctx));
```

Listing 4: Git commit of an adsprpc out-of-bounds access.

Another critical source of information is the source code for any *kernel modifications*, including driver code, which must be released under the GNU General Public License version 2 (GPLv2). This obligation arises from Android's use of the GPLv2 licensed Linux kernel: Any distributed modified versions must also have their corresponding source code made publicly available under the same terms. Here, we exploit this open-source policy to identify n-day driver vulnerabilities.

History Tree Search. Multiple kernel driver source codes are available via git repositories hosted by Google or ODM vendors, such as Qualcomm. For example, Qualcomm's DSP `frpc-adsprpc` kernel driver repository is publicly accessible at https://git.codelinaro.org/clo/la/platform/vendor/qcom/opensource/dsp-kernel.git. While we demonstrate our approach using two security flaws in the `frpc-adsprpc` driver, this method can be applied generically to all publicly available repositories.

Our approach involves searching the entire repository history for keywords that suggest a commit addresses a security flaw. Examples of such keywords include *bug*, *use-after-free*, and *out-of-bounds*. Using these, we identify vulnerabilities in the git repository. One example is commit `2466b` in mid-2024, as shown in Listing 4. This patch introduces a check to ensure that the `sesscount` member variable from the `struct fastrpc_channel_ctx` remains within its intended range of `[0, NUM_SESSIONS-1]`. This patch adds the following check: `VERIFY(err, chan-`

```
commit 3a1e7d811168a32b10171905503d724605064238
Author: DEEPAK SANNAPAREDDY <quic_sdeeredd@quicinc.com>
Date:   Fri Sep 22 16:32:06 2023 +0530

    msm: adsprpc: Handle UAF in process shell memory

    Added flag to indicate memory used
    in process initialization. And, this memory
    would not removed in internal unmap to avoid
    UAF or double free.

    Change-Id: Ie470fe58ac334421d186feb41fa67bd24bb5efea
    Signed-off-by: DEEPAK SANNAPAREDDY

diff --git a/dsp/adsprpc.c b/dsp/adsprpc.c
index 2c28969..43648e9 100644
--- a/dsp/adsprpc.c
+++ b/dsp/adsprpc.c
@@ -4351,6 +4351,8 @@ static int
      fastrpc_init_create_static_process(struct fastrpc_file
      mutex_lock(&fl->map_mutex);
      err = fastrpc_mmap_create(fl, -1, NULL, 0, init->mem,
        init->memlen, ADSP_MMAP_REMOTE_HEAP_ADDR, &mem);
+     if (mem)
+       mem->is_filemap = true;
      mutex_unlock(&fl->map_mutex);
      if (err || (!mem))
        goto bail;
```

Listing 5: Git commit of an adsprpc UAF access.

`>sesscount < NUM_SESSIONS)`. Before, a malicious actor could exploit the vulnerability to perform an Out-Of-Bounds (OOB) write in the `memcpy` function. Exploitation is possible as `chan->session[chan->sesscount]` (aliased as `dup_sess`) would be misinterpreted as `fastrpc_channel_ctx`. OOB writes are a common initial exploit primitive with the potential for system compromise [9, 45, 48, 49, 91].

Another example of an identified n-day vulnerability is the end-2023 Use-After-Free (UAF) flaw involving the `fastrpc_mmap`, as shown in Listing 5. We identified this flaw by searching the history for *UAF*. According to the commit message, this vulnerability can pivot to a Double-Free (DF) scenario, also a robust exploit primitive [45, 49, 86, 91].

5.2 N-Day Analysis

We manually collect publicly available git repositories and bug reports. The repositories are sourced from device OEMs (e.g., Xiaomi and OnePlus), ODMs (e.g., Qualcomm and MediaTek), and Google. The bug reports are obtained from platforms such as Google Project Zero's issue tracker. None of the bug reports referenced the specific patches that fixed the discovered vulnerabilities. It was, therefore, part of our analysis to identify the corresponding patches for the report.

Our analysis involves a two-step process. First, we automatically filter commit messages from these repositories using security-related keywords, as described in Section 5.1. We started with commit messages from 2020 to December 2024 (date of analysis). Then, we manually verified the filtered commits to confirm whether the changes addressed security-critical bugs. Only commits with clear evidence are included in our analysis, e.g., in Listings 4 and 5. Using this approach, we identified 50 security-critical issues: 45 initially from git repositories and 7 from bug reports, with 2 representing duplicates found in both sources. For the analysis, we categorized them based on their exploit capabilities. While several stem from race conditions, their capabilities result in a UAF or an OOB write. The n-day vulnerabilities are classified into four categories (see Table 2):

UAF: Use-after-free access and double-free.

OOB: Out-of-bound write.

Others: Uninitialized variable access, null pointer dereference and denial of service.

ID: Out-of-bound read and information disclosure.

These categories are critical for compromising Android devices. UAF, DF, and OOB write capabilities are particularly notable, as they serve as initial exploit primitives with numerous exploit techniques for system compromise [9, 13, 19, 45, 49, 60, 84, 86, 91]. Other vulnerabilities, such as null pointer dereferences and Denial Of Service (DOS), also offer pathways to root. For example, Jenkins demonstrated an innovative approach to exploiting these issues [28]. Similarly, prior work has shown how to effectively exploit Uninitialized Variables (UV) [11, 38, 47]. Information Disclosure (ID) capabilities are essential in end-to-end exploitation, as demonstrated by prior research [43, 44, 48, 50]. These vulnerabilities allow malicious actors to locate target kernel objects, which most kernel exploits require [26, 27, 50, 63, 65, 66, 78].

Table 2: The capability granted by n-day vulns per driver.

Driver Module	Capability				
	UAF	OOB	Others	ID	Total
`frpc-adsprpc.ko`	12	3	1	3	19
`jpeg-driver.ko`	2	1	0	0	3
`mtk_perf_ioctl.ko`	1	0	4	7	12
`apusys.ko`	0	1	1	1	3
`ged.ko`	1	2	3	3	9
`trusted_mem.ko`	0	2	1	0	3
`xlog.ko`	0	0	1	0	1
	16	9	11	14	50

UAF: Use-after-free and double-free **OOB:** Out-of-bound write
Others: Uninit variable, null pointer deref and denial of service
ID: Out-of-bound read and information disclosure

Table 2 shows the results of our analysis of n-day driver vulnerabilities (excluding GPU drivers), with 50 vulnerabilities identified. Dividing the results into different categories, we observe that vulnerabilities affecting UAF capabilities are the most prevalent, while OOB writes are the least prevalent. We also observe a variation in the number of vulnerabilities identified per driver. We did not find matching git repositories for Samsung's `npu.ko` and `camera.ko`, Xiaomi's `migt.ko` and the `apuext` part of `apusys.ko`. For `mi_log.ko`, we found a repository, but no commits indicating security-related fixes.

Validity of Analysis. We reconstructed n-day vulnerabilities using open-source information from the respective

Table 3: Device firmware categorized by security patch level and release date by OEM vendor.

OEM	2023												2024											Total
	1	2	3	4	5	6	7	8	9	10	11	12	1	2	3	4	5	6	7	8	9	10	11	
Asus	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	0	1	2	6
OnePlus	0	0	3	1	0	0	0	0	1	0	2	0	0	0	0	1	0	1	0	0	0	0	0	9
Oppo	0	0	0	0	0	0	0	0	1	0	0	1	2	0	0	0	0	2	1	0	0	0	0	7
Realme	0	0	0	2	0	0	0	0	0	0	0	1	0	2	1	0	2	0	1	3	3	0	1	16
Vivo	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3	1	0	0	1	2	0	0	7
Xiaomi	0	0	1	1	5	2	3	1	2	2	2	12	12	8	21	10	16	16	26	24	30	19	18	231
Samsung	1	3	4	4	0	2	4	5	2	0	6	12	7	8	15	14	10	24	10	19	12	25	30	217

git repositories. Our reconstruction is based on security experts identifying patches that fix vulnerabilities, including capabilities such as UAF, DF, or OOB writes. However, we do not claim or evaluate whether each specific vulnerability enables direct system compromise. We even argue that limiting the focus solely to vulnerabilities already proven to enable direct system compromise overlooks broader security risks and creates a harmful trend in vulnerability prioritization. The rise of novel exploitation techniques [9, 19, 26, 28, 39, 44–47, 49, 60, 79, 84–86, 88, 90, 91] demonstrates that increasingly weaker exploit primitives can still lead to system compromise despite modern defenses. For example, overwriting one byte with zero [13, 45, 59] has been shown to compromise modern systems. Similarly, null pointer dereferences, which were considered mitigated after the introduction of `mmap_min_addr`, were re-enabled due to a novel kernel exploit technique to compromise recent Linux systems [28]. This shows that even vulnerabilities perceived as low- to no-risk can be leveraged for system compromise.

Given this, we argue that neither security researchers nor vendors should primarily focus their time and effort on determining whether each n-day vulnerability is directly exploitable for system compromise. This, in turn, can lead to poor prioritisation of vulnerabilities, wasting time and resources that could be spent integrating patches. We, therefore, conclude that if an accessible vulnerability falls into a category known to facilitate system compromise, such as UAF, DF, or OOB writes, this should suffice to classify it as critical. Instead, the focus should be on promptly incorporating patches to mitigate these vulnerabilities and prevent their exploitation. However, while we do not demonstrate the exploitability of each vulnerability, we verify the reachability of vulnerable code for a representative subset of vulnerabilities on a real device, supporting n-day exploitability (see Section 6.3).

6 Detecting N-Day Patches in Kernel Drivers

In this section, we perform a large-scale analysis of the inclusion of n-day patches in kernel modules for ODM-specific device drivers. We examine the absence and delay of patches for a subset of the 50 n-day vulnerabilities identified in Section 5.2. Our dataset comprises 493 firmware versions from 131 devices across 7 OEMs, with multiple versions available for devices from Samsung and Xiaomi. For other devices, we use the most recent firmware available.

To assess the susceptibility of devices to n-day vulnerabilities, we evaluate each device based on its most recent available firmware. Specifically, we test whether known vulnerabilities—publicly available before this available firmware release—are still present. We show that 61.4 % of these devices have at least one n-day vulnerability, making them vulnerable to exploitation. Alarmingly, 59.1 % of these devices contain at least one highly critical n-day vulnerability. These highly critical vulnerabilities fall into the UAF or OOB categories. They provide initial exploit primitives with well-researched exploit techniques [9, 13, 19, 45, 49, 60, 84, 86, 91]. We also analyze the patch delay across 131 devices based on their respective security patch levels. Our results show significant delays in patch integration, with delays up to 830 d. These results demonstrate that our identified drivers are susceptible targets for malicious actors, satisfying **C3**.

To perform this large-scale analysis, we present a semi-automatic approach for detecting patch presence for 21 of the 50 n-day driver vulnerabilities (see Section 6.1). Using this approach, we analyze compiled kernel modules extracted from device firmwares (see Section 6.2). Lastly, we demonstrate on a representative subset of these vulnerabilities that they are triggerable on a real device (see Section 6.3).

Firmware Versions. We have a total of 493 firmware versions from 131 devices across 7 OEM vendors. For Xiaomi and Samsung, our collection contains the latest firmware version for multiple devices as well as older versions. For the other OEM vendors (i.e., Oppo, OnePlus, Asus, Realme, and Vivo), we analyze the most recent firmwares available online, ranging between early-2023 and end-2024. Table 3 shows the amount of firmwares for each security patch level and OEM.

6.1 Patch Detection Approach

We demonstrate a semi-automated approach for detecting security patches in kernel drivers by analyzing code modifications in the compiled kernel modules. We exclude patches that fall outside our analytical framework's strategies, which

focus on two primary methods: symbol-based detection and control-flow analysis through decompilation.

Symbol-Based Detection. We leverage the fact that security patches frequently introduce symbols. These could, e.g., be references to global functions, global variables, or unique string artifacts for error messages and diagnostic output. For globally accessible symbols, our approach analyzes whether the symbols introduced in the source patch are present in the compiled target driver. We exclude inlined function symbols, as these may not result in detectable changes in the compiled binary. To eliminate false negatives, we only consider cases where these symbols are also present in the most recent version of the kernel driver. String artifacts provide an additional possibility for patch detection. Security patches frequently include unique strings, such as warnings or diagnostic messages (see Listing 4), that can be used as identifiers in the binary representation of the driver. We exclude cases where introduced strings are not unique, e.g., because they had been used in other places before the patch already.

On symbol absent, further manual verification is required to rule out false negatives, involving two key considerations: kernel driver configurability and code evolution. Kernel drivers often implement configurations that selectively exclude code segments, requiring manual analysis to verify whether patch-affected code exists. Additionally, subsequent or custom commits may modify or replace strings the patch introduces, potentially obscuring its presence. Our approach accounts for these changes to ensure accurate patch detection.

Control-Flow Analysis. Security patches frequently implement modifications to program control flow [35, 93], typically through additional conditional logic, as demonstrated in Listing 5. Our control-flow analytical approach focuses on identifying patches that modify program logic on patch-modified functions through a two-phase process. Initially, we perform differential analysis of assembly code across sequential driver versions. The absence of assembly-level differences between versions indicates patch exclusion within that interval. For versions exhibiting assembly modifications, we employ Ghidra to perform decompilation. Although the decompilation output does not perfectly match the original code, it enables the identification of control-flow modifications through comparative manual analysis. Combining both approaches allows for efficient manual verification while not degrading output performance, especially since most changes in driver functions are due to security issues, as observed.

Future Work. While we used a tailored approach to detect patches in ODM-specific drivers, prior work [35, 93] focused on pre-GKI kernel images. Notably, PDiff [35] did not release their approach as open source, but Fiber [93] or similar approaches, such as those using angr [67], could be adapted to detect the absence of patches. However, Fiber was originally designed for kernel images rather than kernel modules, and adapting it to our methodology would require significant engineering effort. Fiber's evaluation was also limited to 11 kernel images using custom execution allowlists. To meet the requirements of our dataset, which includes 493 firmwares, each containing 1 to 5 relevant kernel modules, we would need to significantly extend Fiber's capabilities for efficient patch detection. Given these challenges, we suggest extending Fiber or developing a similar framework as future work.

Table 4: Devices' n-day susceptibility to vulnerabilities known as of their most recent firmware release. **All Device Analyzed** includes all studied devices with the most firmware versions, while **Devices with Target Drivers** refers to those having hardware support for at least one of the target drivers.

OEM	All Devices Analyzed		Devices with Target Drivers	
	Crit Vuln	Any Vuln	Crit Vuln	Any Vuln
	%	%	%	%
Samsung	45.5	45.5	74.1	74.1
Xiaomi	67.3	71.4	75.0	79.5
Asus	75.0	100.0	75.0	100.0
Realme	56.2	62.5	56.2	62.5
Vivo	40.0	40.0	40.0	40.0
Oppo	42.9	42.9	42.9	42.9
OnePlus	85.7	85.7	85.7	85.7

6.2 Large-Scale Analysis on Patch Inclusion

By integrating symbol-based detection and control-flow analysis, our methodology ensures the identification of security-related patches. To evaluate the effectiveness of our approach, we select 21 n-day vulnerabilities from the full set of 50 identified, focusing on vulnerabilities in the DSP, JPEG, and GED kernel drivers, such as Listings 4, 5 and 6 to 10. These vulnerabilities include highly critical ones (i.e., 9 UAF and 6 OOB) as well as moderately critical ones (i.e., 1 Others and 5 ID), representing a balanced mix of different severity levels. Our main goals are: to quantify how many devices—running their respective newest available firmware versions—remain susceptible to these n-day vulnerabilities and assess security-relevant patch delays.

Device Susceptibility. We assess device susceptibility under two conditions. First, we analyze the most recent firmware available for all 131 Android devices in our dataset (see Table 3) to determine what fraction of the 21 n-day vulnerabilities each device is still vulnerable to. Second, we repeat this only for devices with hardware support for at least one of the target kernel drivers identified as potentially vulnerable.

Table 4 shows susceptibility rates, separating results for all devices (**All Devices Analyzed**) and those with hardware support of at least one target driver (**Devices with Target Drivers**). In both cases, we observe widespread susceptibility to highly and moderately critical n-day vulnerabilities. For instance, 45.5 % of Samsung devices and 71.4 % of Xiaomi devices were found to be vulnerable to at least one n-day vulnerability. Aggregated across all devices, 59.1 % were vulnerable to at least one highly critical vulnerability,

Table 5: Lower bound for average n-day patch delays in years across device OEM vendors and vulnerability categories.

OEM	Average Delay Time (Lower Bound)			
	UAF	OOB	Others	ID
Samsung	0.32 y±0.4	0.40 y±0.1	0.32 y±0.0	0.79 y±0.7
Xiaomi	0.56 y±0.4	0.70 y±0.9		0.88 y±0.6

Table 6: Lower bound for average n-day patch delays in years across device ODMs and vulnerability categories.

ODM	Average Delay Time (Lower Bound)			
	UAF	OOB	Others	ID
Qualcomm	0.38 y±0.4	0.23 y±0.2	0.32 y±0.0	0.65 y±0.6
MediaTek	0.71 y±0.5	2.15 y±0.1		2.00 y±0.3

while 61.4 % were vulnerable to at least one of any severity. Susceptibility rates for Samsung and Xiaomi appear lower primarily due to the lack of hardware support for the drivers we analyzed. When excluding devices without the analyzed hardware—like Samsung's NPU—susceptibility rates rise to 74.1 % for Samsung and 79.5 % for Xiaomi.

Crucially, while Table 4 shows patching trends, it does not support a direct comparison between lineage-providing OEMs (i.e., Xiaomi and Samsung) and others, as, e.g., many 2024 vulnerabilities cannot be tested on non-lineage vendors that only publicly provide older firmwares.

OEM Breakdown. Susceptibility is often not limited to a single vulnerability. We find a correlation between being affected by one vulnerability and being affected by multiple. For example, 29.5 % of Samsung devices and 49 % of Xiaomi devices were vulnerable to three or more n-day vulnerabilities.

Takeaway 1

If a device is susceptible to 1 n-day vulnerability, it is likely susceptible to multiple vulnerabilities.

We also observe a trend in patch behavior across OEMs, with Xiaomi standing out. Active patching of vulnerabilities in Xiaomi devices is limited, with only a few instances observed that patch the identified n-day vulnerabilities. In many cases, when the earliest firmware version of a device is found to be susceptible to n-day vulnerabilities, subsequent firmware versions tend to remain susceptible. This pattern suggests that security flaws are often addressed indirectly through the release of newer Android devices with updated kernel driver versions that include the necessary patches rather than through firmware updates for existing devices. While this approach is more prominent in Xiaomi's practices, similar tendencies are also observed, albeit notably lesser, in Samsung's handling of such vulnerabilities.

Takeaway 2

Security-related flaws are likely addressed through new device releases than through updates to existing devices.

Patch Delays. We define a patch as either integrating a fix into existing software or entirely replacing the affected software with a non-susceptible version. Our analysis reveals variability in how patches propagate across firmware versions. Table 5 quantifies the time gap between the released commit date of a patch and the last analyzed firmware version where the patch was missing. The precision of these results depends on the granularity of our firmware dataset, which varies between OEM vendors. Our dataset contains temporal gaps of months. Patch integration could have occurred between the release of the last unpatched and the first patched firmware in our dataset. Our results are, therefore, conservative estimates for the patch integration delay. Specifically, some patches are missing from the latest firmware releases at the time of our analysis. In such cases, we conservatively estimate the delay metrics by assuming that the patch will be included in the next firmware release, which is a lower bound. Hence, our delay measurements will likely underestimate actual patch delays, which may be higher.

Table 5 highlights variations in patch delays across OEMs and vulnerability types. Samsung generally patches security-critical flaws faster than Xiaomi. Among vulnerability types, OOB and Others are patched the quickest, followed by UAF, while ID takes the longest. This trend reflects the perception that information disclosure vulnerabilities are less critical than UAF or OOB. However, UAF and OOB vulnerabilities can still take over 500 d to patch, with ID exceeding 800 d. On average, OOB vulnerabilities are patched more quickly than UAF, which are patched faster than ID.

ODM Breakdown. In addition to analyzing patch delays solely by OEM and vulnerability type, we further break down the results by ODM chipset. Samsung's Exynos-based devices are excluded from this analysis due to the lack of publicly available source repositories. Table 6 presents the results of this ODM-based breakdown, revealing notable differences in patching delays between Qualcomm- and MediaTek-based chipsets. Compared to their Qualcomm counterparts, MediaTek devices consistently exhibit longer patch delays of over two times across OEMs and multiple vulnerability classes. For example, for UAF vulnerabilities, the average patch delay increases from approximately 4 months for Qualcomm to over 8 months for MediaTek.

Takeaway 3

While the patch delay varies by OEM, ODM, and vulnerability, some devices experience delays of over a year.

6.3 Representative Subset of N-Day Vulns

To support the claim that n-day vulnerabilities remain exploitable, we demonstrate that a representative subset can be reliably triggered on a real Android device. Developing Proof-of-Concept (PoC)s is a non-trivial task requiring substantial

time, device access, and in-depth driver-specific expertise, even for experienced analysts like those at Google Project Zero [23, 26]. Given this complexity, we focus the following analysis on a single device driver to ensure feasibility. We target `frpc-adsprpc.ko`, which accounts for most identified vulnerabilities. Since we apply the same patch-based approach across all affected modules, dynamically validating one representative driver shows reachability and supports the generalizability of our approach.

We select five vulnerabilities from this driver with their git patch commit messages shown in Listings 6 to 10. These five vulnerabilities represent a range of bug classes: one OOB read (**vuln1**), three UAF issues (**vuln2-4**), and one information disclosure (**vuln5**). We evaluate these across 29 module versions covering various release dates and 13 Android devices from three major OEMs (Samsung, Xiaomi, and Asus), providing a representative subset of both OEMs and n-day exposure timelines. Our findings (see Section 6.3.1) show that on Samsung devices, the vulnerabilities remain n-day exploitable for 0 to 7 months. In contrast, on all tested Xiaomi and Asus devices, multiple vulnerabilities remained exploitable even at the time of analysis. The results for the representative subset align with the results from our patch detection approach, supporting the validity of our static determination (see Section 6.3.2).

We use a rooted Samsung Galaxy S23 equipped with the Qualcomm SM8550-AC Snapdragon 8 Gen 2 chipset for testing. Even after device rooting, most partitions (including the one containing kernel drivers) are mounted using the Enhanced Read-Only File System (EROFS), preventing direct modification. To circumvent this, we flash TWRP/RO2RW images and remount the `vendor_dlkm` partition as writable, enabling us to replace the `frpc-adsprpc.ko` driver with a test version. Using this setup, we test driver versions from 13 Android phones, spanning multiple timestamps and three OEMs. All 13 devices use the same chipset as our test device, ensuring compatibility. We manually confirm that substituting the driver module from another OEM with the same chipset does not alter functionality. This allows us to test different versions while keeping the engineering effort reasonable.

Triggering the vulnerabilities produces two observable effects: the UAF issues and the OOB read cause a crash, while the information disclosure leaks a kernel pointer. We use publicly available PoCs where possible (e.g., for **vuln1** [32]) or develop one (e.g., for **vuln5** [31]) to trigger each vulnerability.

6.3.1 Analysis

We present our findings in Table 7. We begin by testing 10 versions of `frpc-adsprpc.ko` from the original Galaxy S23, covering firmware releases available between March 2024 (the earliest relevant commit) and December 2024 (the time of analysis). We find that the October 2024 release is the first to include all five patches that mitigate the tested vulnerabilities. All vulnerabilities are tested for S23 original drivers. However, we omit **vuln3** in broader tests on devices other than the S23 due to its long triggering time (more than 12 hours).

Next, we test the September and October 2024 driver versions across 4 additional Samsung models (Galaxy S23+, S23 Ultra, Z Flip5, and Z Fold5). Consistent with the S23 results, the October release contains the full set of patches. Patch delays for Samsung devices range from 0 to 7 months, depending on the vulnerability.

We then test the November and December 2024 releases for six Xiaomi models (MIX Fold 3, Redmi K70, POCO F6 Pro, Xiaomi 13 Ultra, 13 Pro, and 13) and the November 2024 release for two ASUS models (ROG Phone 7 and 7 Ultimate). All of these most recent releases remain vulnerable to **vuln2/5**, while four are also vulnerable to **vuln1/4**. Patch delays vary between 0 and over 9 months, depending on the device and specific vulnerability.

Takeaway 4

A PoC for an n-day vulnerability in ODM-maintained drivers can be reused across OEMs and timeframes.

6.3.2 Validity Check of Patch Detection

We have two sets of patch inclusion analysis results: the large, statically determined set described in Section 6.2, and the smaller, dynamically determined subset discussed in Section 6.3. We now validate the dynamically determined subset by comparing it against the statically determined results, confirming consistency between them.

7 Discussion and Related Work

This section discusses the security implications and validity of our findings, as well as related work.

Security Implications. Our analysis reveals that modern Android devices have a significant kernel attack surface reachable from untrusted contexts, comprising kernel device drivers. Many of these drivers expose known n-day vulnerabilities for long periods of time, allowing malicious actors to bypass the effort of developing zero-day exploits. Instead, they can exploit these n-day vulnerabilities to compromise devices. Crucially, as a device is only as secure as its weakest point, our research highlights that device drivers represent this weakest link in current Android versions. A single pathway to root access is sufficient to fully compromise Android devices.

Takeaway 5

Malicious actors can exploit n-day vulnerabilities, reducing reliance on time-consuming zero-day development.

Validity of our Results. Our findings are mostly derived from static analysis. To ensure consistency with real-world scenarios, we incorporated dynamic testing and manual verification throughout. While we demonstrate n-day triggering for 5 vulnerabilities in Qualcomm-supplied DSP drivers,

our evaluation does not include full end-to-end exploitation. Furthermore, static analysis has inherent limitations, such as missing dynamic behaviors, and our evaluation focuses on a subset of drivers and devices. As a result, there may be unknown barriers to triggering the vulnerabilities and achieving full exploitation. The reported numbers should, therefore, be interpreted as estimates of real-world exploitability.

Patch Detection and Propagation. Prior work [35, 42, 93] has demonstrated methods for detecting patches in kernel binaries by, e.g., deriving patch's signatures and testing them against the kernel binary. Numerous studies have explored solutions to mitigate the effects of delayed patch integration. Wang et al. [80] proposed temporary patch integration, while Chen et al. [10] and Xu et al. [89] introduced hot patch techniques to mitigate vulnerabilities dynamically. Talebi et al. [77] prevented harmful side effects of vulnerable code through syscall instrumentation. Another security-related problem is that while patches are available, vendors are reluctant to apply them. Hence, other research [51] has focused on faster and more correct patch propagation.

Patch and Defense Integration. The deployment of security updates and defenses in Android systems has been a focus of various studies. Wu et al. [81] highlighted that most issues in the Android Security Bulletin (ASB) originate from native code, while Farhang et al. [15] observed that kernel-related CVEs experience the longest delays in propagating to vendor ASBs. This delay creates a window for attackers to exploit vulnerabilities before patches reach end users. Jones et al. [37] and Zhang et al. [95] quantified the time lag, reporting delays of weeks to months for Android security updates. Acar et al. [1] revealed significant fragmentation in Android's security update ecosystem, with inconsistent and delayed patch rollouts across devices, vendors, and regions. Maar et al. [48] recently analyzed the challenges of integrating mainline kernel defenses against n-day exploitation. Most of these studies focused on Android versions that predate the GKI initiative, which was intended to address kernel patch delays for good. However, we show that these delays remain a threat to device security as vendors struggle to integrate patches in those parts of the kernel.

Security Analysis on Android. Google Project Zero has been tracking zero-day exploits targeting Android since 2019 [69]. Their annual reviews [69, 71, 74, 76] analyze trends in malicious actor behavior to enhance Android security. These reports emphasize the critical role of timely patch deployment and defense integration in mitigating in-the-wild exploitation [61]. Other research groups, such as the Threat Analysis Group [75], GitHub Security Lab [55–58], Zero Day Engineering [14], Blue Frost Security [65, 66], Amnesty International's Security Lab [23, 24, 61], and Citizen Lab [40, 41, 53, 61], also analyze exploitation trends.

Android Driver Security. Exploiting vulnerabilities in kernel drivers, particularly the GPU, has been a key focus of recent research [8, 14, 16–18, 56–58, 73, 75, 76, 84, 87]. Collaboration between Google, Android, and ARM has contributed to advances in vulnerability detection, mitigation, and hardening [87]. NPU drivers have also been targeted [55, 62, 83, 94], although such exploits have predominantly focused on Samsung devices due to the unclear adoption of similar vulnerabilities in Qualcomm-based devices [55].

Research on DSP-related kernel drivers remains sparse [14], but recent and concurrent exploits [23, 33] show that these drivers were actively exploited to compromise Android devices. Malicious actors exploited these drivers to install spyware, such as NoviSpy, which specifically targets end users in the wild [23]. These findings underscore the relevancy of our work: *the exploitability of device drivers is known to malicious actors, so it is imperative that research catches up*.

Most recently, concurrent work by Amnesty International's Security Lab [24] demonstrated that kernel device drivers remain a primary attack vector for Android compromise. In particular, malicious actors deployed zero-day exploits against Android USB kernel drivers observed in-the-wild. Similar to the 2024 DSP zero-day exploits [23], malicious actors then installed NoviSpy spyware for surveillance.

While some studies have examined vulnerabilities in drivers accessible from trusted contexts [36, 82], the pervasive threat posed by drivers accessible from untrusted contexts remains largely unexplored. To the best of our knowledge, Jenkins has done the closest analysis of the kernel attack surface from untrusted security contexts [29]. Jenkins analyzed the attack surface for 3 devices, i.e., Google Pixel 7, Xiaomi 11T, and Asus ROG 6D, and presented multiple zero-day vulnerabilities, demonstrating that security research on Android drivers is sparse. We, on the other hand, performed a large-scale analysis of 131 devices and discovered a large number of drivers that are accessible and, worse, exploitable with n-day vulnerabilities from untrusted contexts.

8 Conclusion

Prior compromises of Android devices often relied on exploit chains targeting GPU kernel drivers or higher user-space privileges before targeting the kernel. In this paper, we comprehensively analyzed the kernel attack surface exposed to untrusted security contexts. Our analysis reveals that this attack surface is significantly larger than previously known, comprising multiple device drivers. Within these drivers, we identified vulnerabilities that remain unpatched for extended periods or were still unpatched at the time of our analysis. Specifically, 59.1 % of recent Android devices were vulnerable to highly critical n-day exploits, with 61.4 % affected by any vulnerability. These unpatched vulnerabilities present an ideal target for malicious actors, as they eliminate the need to invest substantial time and effort into developing zero-day exploits. This critical state of Android kernel security highlights the need for urgent action to enhance patch management and improve overall security.

9 Acknowledgements

We thank the anonymous reviewer and shepherd for their valuable feedback. We also thank Jann Horn and Seth Jenkins for their help with triggering the PoCs, and the MediaTek Product Security Team for identifying an error we subsequently fixed. This research was funded in whole or in part by the Austrian Research Promotion Agency (FFG) via the SEIZE and AWARE project (FFG grant numbers 888087 and 915106). Any opinions, findings, conclusions, or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding parties.

10 Ethics Considerations

We responsibly disclosed our findings to all affected parties, including OEMs (Samsung, Xiaomi, Asus, Realme, OnePlus, Oppo, and Vivo), ODMs (Samsung, Qualcomm, and MediaTek), and Google.

- Asus, Realme, and MediaTek acknowledged our findings.
- Samsung acknowledged the issues and has initiated patching for affected devices. We had a follow-up meeting focused on improving Android driver security.
- Google responded by stating that the issues fall outside their scope of enforcement and directed us to report the findings to OEMs directly.

Following best practices from Google Project Zero, we gave all participants more than 90 days to address issues, with a 30-day grace period. This timeline left plenty of buffer time for the earliest possible release date of this paper, allowing all participants ample opportunity to develop and implement comprehensive solutions before public release.

Moreover, we believe revealing these findings is crucial to demonstrating how malicious actors can exploit the Android environment. This underscores the need for urgent action to improve overall Android security by addressing patch delays, prioritizing device updates, and securing kernel attack surfaces. We strongly believe that publishing our findings will improve the security of Android devices in the long term and is, therefore, the most ethical course of action.

We are committed to an ethical approach that balances responsible research with potential security improvements. Our research relies exclusively on publicly available firmware images obtained from multiple sources, including official OEM/ODM vendor repositories and third-party providers. While we recognize that methodological precedent alone cannot justify research ethics, prior work [1,37,48,95] reinforces our belief in the validity of analyzing publicly available data. We have not analyzed how the firmwares have been obtained by the third-party providers.

All dynamic testing was conducted in a controlled laboratory environment using dedicated research devices, further ensuring the integrity and safety of our investigation.

11 Open Science

While we aim to make all datasets, crawling tools, and analysis scripts open source, doing so poses a risk of misuse by malicious actors, as observed in the past [27,53,68,70,75,76]. Consequently, we do not recommend open-sourcing these resources. However, if the USENIX committee holds a different opinion, we will share all our findings and tools accordingly.

References

[1] Abbas Acar, Güliz Seray Tuncay, Esteban Luques, Harun Oz, Ahmet Aris, and Selcuk Uluagac. 50 Shades of Support: A Device-Centric Analysis of Android Security Updates. In *NDSS*, 2024.

[2] Android. Application Sandbox, 2021. URL: https://source.android.com/security/app-sandbox.

[3] Android. Build SELinux policy, 2024. URL: https://source.android.com/docs/security/features/selinux/build.

[4] Android. Generic Kernel Image (GKI) project, 2024. URL: https://source.android.com/docs/core/architecture/kernel/generic-kernel-image.

[5] AppBrain. Top manufacturers, 2024. accessed: 31.12.2024. URL: https://web.archive.org/web/20241230133601/https://www.appbrain.com/stats/top-manufacturers.

[6] Brandon Azad. A survey of recent ios kernel exploits, 2020. URL: https://googleprojectzero.blogspot.com/2020/06/a-survey-of-recent-ios-kernel-exploits.html.

[7] Jia-Ju Bai, Julia Lawall, Qiu-Liang Chen, and Shi-Min Hu. Effective Static Analysis of Concurrency Use-After-Free Bugs in Linux Device Drivers. In *USENIX ATC*, 2019.

[8] Ian Beer. Mind the Gap, 2022. URL: https://googleprojectzero.blogspot.com/2022/11/.

[9] Weiteng Chen, Xiaochen Zou, Guoren Li, and Zhiyun Qian. KOOBE: Towards Facilitating Exploit Generation of Kernel Out-Of-Bounds Write Vulnerabilities. In *USENIX Security*, 2020.

[10] Yue Chen, Yulong Zhang, Zhi Wang, Liangzhao Xia, Chenfu Bao, and Tao Wei. Adaptive Android Kernel Live Patching. In *USENIX Security*, 2017.

[11] Haehyun Cho, Jinbum Park, Joonwon Kang, Tiffany Bao, Ruoyu Wang, Yan Shoshitaishvili, Adam Doupé, and Gail-Joon Ahn. Exploiting Uses of Uninitialized

Stack Variables in Linux Kernels to Leak Kernel Pointers. In *WOOT*, 2020.

[12] Chromium. PowerVR GPU - Kernel heap OOB write in RGXFWChangeOSidPriority - CVE-2024-23698, 2024. URL: https://apvi.issues.chromium.org/issues/42420036.

[13] Devil. CoRJail: From Null Byte Overflow To Docker Escape Exploiting poll_list Objects In The Linux Kernel, 2022. URL: https://syst3mfailure.io/corjail/.

[14] Alisa Esage. Deep Dive: Qualcomm MSM Linux Kernel & ARM Mali GPU 0-day Exploit Attacks of October 2023, 2023. URL: https://zerodayengineering.com/insights/qualcomm-msm-arm-mali-0days.html.

[15] Sadegh Farhang, Mehmet Bahadir Kirdan, Aron Laszka, and Jens Grossklags. An empirical study of android security bulletins in different vendors. In *WWW*, 2020.

[16] Guang Gong. TiYunZong: An Exploit Chain to Remotely Root Modern Android Devices, 2020. URL: https://github.com/secmob/TiYunZong-An-Exploit-Chain-to-Remotely-Root-Modern-Android-Devices/blob/master/us-20-Gong-TiYunZong-An-Exploit-Chain-to-Remotely-Root-Modern-Android-Devices-wp.pdf.

[17] Xiling Gong, Xuan Xing, and Eugene Rodionov. The Way to Android Root: Exploiting Your GPU On Smartphone, 2024. URL: https://i.blackhat.com/BH-US-24/Presentations/REVISED02-US24-Gong-The-Way-to-Android-Root-Wednesday.pdf.

[18] Ben Hawkes. Attacking the Qualcomm Adreno GPU, 2020. URL: https://googleprojectzero.blogspot.com/2020/09/attacking-qualcomm-adreno-gpu.html.

[19] Jann Horn. How a simple Linux kernel memory corruption bug can lead to complete system compromise, 2021. URL: https://googleprojectzero.blogspot.com/2021/10/how-simple-linux-kernel-memory.html.

[20] Jann Horn. CVE-2022-22706 / CVE-2021-39793: Mali GPU driver makes read-only imported pages host-writable, 2022. URL: https://googleprojectzero.github.io/0days-in-the-wild//0day-RCAs/2021/CVE-2021-39793.html.

[21] Jann Horn. adsprpc: refcount leak leading to UAF in fastrpc_get_process_gids, 2024. URL: https://project-zero.issues.chromium.org/issues/42451711.

[22] Amnesty International. Forensic Methodology Report: How to catch NSO Group's Pegasus, 2021. URL: https://www.amnesty.org/en/latest/research/2021/07/forensic-methodology-report-how-to-catch-nso-groups-pegasus/.

[23] Amnesty International. "A Digital Prison": Surveillance and the suppression of civil society in Serbia, 2024. URL: https://securitylab.amnesty.org/latest/2024/12/a-digital-prison-surveillance-and-the-suppression-of-civil-society-in-serbia/.

[24] Amnesty International. Cellebrite zero-day exploit used to target phone of Serbian student activist, 2025. URL: https://securitylab.amnesty.org/latest/2025/02/cellebrite-zero-day-exploit-used-to-target-phone-of-serbian-student-activist/.

[25] Amnesty International. Europe: Paragon attacks highlight Europe's growing spyware crisis, 2025. URL: https://www.amnesty.org/en/latest/news/2025/03/europe-paragon-attacks-highlight-europes-growing-spyware-crisis/.

[26] Seth Jenkins. Exploiting CVE-2022-42703 - Bringing back the stack attack, 2022. URL: https://googleprojectzero.blogspot.com/2022/12/exploiting-CVE-2022-42703-bringing-back-the-stack-attack.html.

[27] Seth Jenkins. Analyzing a Modern In-the-wild Android Exploit, 2023. URL: https://googleprojectzero.blogspot.com/2023/09/analyzing-modern-in-wild-android-exploit.html.

[28] Seth Jenkins. Exploiting null-dereferences in the Linux kernel, 2023. URL: https://googleprojectzero.blogspot.com/2023/01/exploiting-null-dereferences-in-linux.html.

[29] Seth Jenkins. Driving forward in Android drivers, 2024. URL: https://googleprojectzero.blogspot.com/2024/06/driving-forward-in-android-drivers.html.

[30] Seth Jenkins. FASTRPC_ATTR_KEEP_MAP logic bug allows fastrpc_internal_munmap_fd to concurrently free in-use mappings leading to UAF, 2024. URL: https://project-zero.issues.chromium.org/issues/42451725.

[31] Seth Jenkins. Incorrect searching algorithm in fastrpc_mmap_find leads to kernel address space

info leak, 2024. URL: https://project-zero.issues.chromium.org/issues/42451713.

[32] Seth Jenkins. is_compat flag in adsprpc driver leads to access of userland provided addresses as kernel pointers, 2024. URL: https://project-zero.issues.chromium.org/issues/42451710.

[33] Seth Jenkins. The Qualcomm DSP Driver - Unexpectedly Excavating an Exploit, 2024. URL: https://googleprojectzero.blogspot.com/2024/12/qualcomm-dsp-driver-unexpectedly-excavating-exploit.html.

[34] Seth Jenkins. UAF race of global maps in fastrpc_mmap_create (and epilogue functions), 2024. URL: https://project-zero.issues.chromium.org/issues/42451715.

[35] Zheyue Jiang, Yuan Zhang, Jun Xu, Qi Wen, Zhenghe Wang, Xiaohan Zhang, Xinyu Xing, Min Yang, and Zhemin Yang. PDiff: Semantic-Based Patch Presence Testing for Downstream Kernels. In *CCS*, 2020.

[36] Xingyu Jin and Clement Lecigene. CVE-2024-44068: Samsung m2m1shot_scaler0 device driver page use-after-free in Android, 2024. URL: https://googleprojectzero.github.io/0days-in-the-wild/0day-RCAs/2024/CVE-2024-44068.html.

[37] Kailani R. Jones, Ting-Fang Yen, Sathya Chandran Sundaramurthy, and Alexandru G. Bardas. Deploying android security updates: an extensive study involving manufacturers, carriers, and end users. In *CCS*, 2020.

[38] Max Kellermann. The Dirty Pipe Vulnerability, 2022. URL: https://dirtypipe.cm4all.com/.

[39] Vasileios P Kemerlis, Michalis Polychronakis, and Angelos D Keromytis. ret2dir: Rethinking kernel isolation. In *USENIX Security*, 2014.

[40] Citizen Lab. ForcedEntry NSO Group iMessage Zero-Click Exploit Captured in the Wild, 2021. URL: https://citizenlab.ca/2021/09/forcedentry-nso-group-imessage-zero-click-exploit-captured-in-the-wild/.

[41] Citizen Lab. Blastpass NSO Group iPhone Zero-Click, Zero-Day Exploit Captured in the Wild, 2023. URL: https://citizenlab.ca/2023/09/blastpass-nso-group-iphone-zero-click-zero-day-exploit-captured-in-the-wild/.

[42] Jakob Lell and Karsten Nohl. Mind the Gap - Uncovering the Android patch gap through binary-only patch analysis, 2018. URL: https://conference.hitb.org/hitbsecconf2018ams/materials/D2T1%20-%20Karsten%20Nohl%20&%20Jakob%20Lell%20-%20Uncovering%20the%20Android%20Patch%20Gap%20Through%20Binary-Only%20Patch%20Level%20Analysis.pdf.

[43] Zhengchuan Liang, Xiaochen Zou, Chengyu Song, and Zhiyun Qian. K-LEAK: Towards Automating the Generation of Multi-Step Infoleak Exploits against the Linux Kernel. In *NDSS*, 2024.

[44] Zhenpeng Lin, Yueqi Chen, Yuhang Wu, Dongliang Mu, Chensheng Yu, Xinyu Xing, and Kang Li. GREBE: Unveiling Exploitation Potential for Linux Kernel Bugs. In *S&P*, 2022.

[45] Zhenpeng Lin, Yuhang Wu, and Xinyu Xing. DirtyCred: Escalating Privilege in Linux Kernel. In *CCS*, 2022.

[46] Zhenpeng Lin, Xinyu Xing, Zhaofeng Chen, and Kang Li. Bad io_uring: A New Era of Rooting for Android, 2023. URL: https://i.blackhat.com/BH-US-23/Presentations/US-23-Lin-bad_io_uring.pdf.

[47] Kangjie Lu, Marie-Therese Walter, David Pfaff, Stefan Nümberger, Wenke Lee, and Michael Backes. Unleashing Use-Before-Initialization Vulnerabilities in the Linux Kernel Using Targeted Stack Spraying. In *NDSS*, 2017.

[48] Lukas Maar, Florian Draschbacher, Lukas Lamster, and Stefan Mangard. Defects-in-Depth: Analyzing the Integration of Effective Defenses against One-Day Exploits in Android Kernels. In *USENIX Security*, 2024.

[49] Lukas Maar, Stefan Gast, Martin Unterguggenberger, Mathias Oberhuber, and Stefan Mangard. SLUBStick: Arbitrary Memory Writes through Practical Software Cross-Cache Attacks within the Linux Kernel. In *USENIX Security*, 2024.

[50] Lukas Maar, Lukas Giner, Daniel Gruss, and Stefan Mangard. When Good Kernel Defenses Go Bad: Reliable and Stable Kernel Exploits via Defense-Amplified TLB Side-Channel Leaks. In *USENIX Security*, 2025.

[51] Aravind Machiry, Nilo Redini, Eric Camellini, Christopher Kruegel, and Giovanni Vigna. SPIDER: Enabling Fast Patch Propagation in Related Software Repositories. In *S&P*, 2020.

[52] Aravind Machiry, Chad Spensky, Jake Corina, Nick Stephens, Christopher Kruegel, and Giovanni Vigna. DR. CHECKER: A Soundy Analysis for Linux Kernel Drivers. In *USENIX Security*, 2017.

[53] Bill Marczak, Adam Hulcoop, Etienne Maynier, Bahr Abdul Razzak, Masashi Crete-Nishihata, John

Scott-Railton, and Ron Deibert. Missing Link Tibetan Groups Targeted with 1-Click Mobile Exploits, 2019. URL: https://citizenlab.ca/2019/09/poison-carp-tibetan-groups-targeted-with-1-click-mobile-exploits/.

[54] Rene Mayrhofer, Jeff Vander Stoep, Chad Brubaker, Dianne Hackborn, Bram Bonné, Güliz Seray Tuncay, Roger Piqueras Jover, and Michael Specter. The Android Platform Security Model (2023). *arXiv:1904.05572*, 2024.

[55] Man Yue Mo. Fall of the machines: Exploiting the Qualcomm NPU (neural processing unit) kernel driver, 2021. URL: https://github.blog/security/vulnerability-research/fall-of-the-machines-exploiting-the-qualcomm-npu-neural-processing-unit-kernel-driver/.

[56] Man Yue Mo. One day short of a full chain: Part 1 - Android Kernel arbitrary code execution, 2021. URL: https://securitylab.github.com/research/one_day_short_of_a_fullchain_android/.

[57] Man Yue Mo. Corrupting memory without memory corruption, 2022. URL: https://github.blog/security/vulnerability-research/corrupting-memory-without-memory-corruption/.

[58] Man Yue Mo. Gaining kernel code execution on an MTE-enabled Pixel 8, 2024. URL: https://github.blog/security/vulnerability-research/gaining-kernel-code-execution-on-an-mte-enabled-pixel-8/.

[59] Andy Nguyen. CVE-2021-22555: Turning x00x00 into 10000$, 2021. URL: https://google.github.io/security-research/pocs/linux/cve-2021-22555/writeup.html.

[60] Lau Notselwyn. Flipping Pages: An analysis of a new Linux vulnerability in nf_tables and hardened exploitation techniques, 2024. URL: https://pwning.tech/nftables/.

[61] Donncha O'Cearbhaill and Bill Marczak. Exploit archaeology a forensic history of in the wild NSO Group exploits. In *Virus Bulletin Conference*, 2022.

[62] Maxime Peterlin. Reversing and Exploiting Samsung's Neural Processing Unit, 2021. URL: https://blog.longterm.io/samsung_npu.html.

[63] Alexander Popov. Four Bytes of Power: Exploiting CVE-2021-26708 in the Linux kernel, 2021. URL: https://a13xp0p0v.github.io/2021/02/09/CVE-2021-26708.html.

[64] Andrea Possemato, Simone Aonzo, Davide Balzarotti, and Yanick Fratantonio. Trust, but verify: A longitudinal analysis of android oem compliance and customization. In *S&P*, 2021.

[65] Eloi Sanfelix. A bug collision tale, 2020. URL: https://labs.bluefrostsecurity.de/files/OffensiveCon2020_bug_collision_tale.pdf.

[66] Blue Frost Security. Exploiting CVE-2020-0041 - Part 2: Escalating to root, 2020. URL: https://labs.bluefrostsecurity.de/blog/2020/04/08/cve-2020-0041-part-2-escalating-to-root/.

[67] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *S&P*, 2016.

[68] Maddie Stone. Bad Binder: Android In-The-Wild Exploit, 2019. URL: https://googleprojectzero.blogspot.com/2019/11/bad-binder-android-in-wild-exploit.html.

[69] Maddie Stone. Detection Deficit: A Year in Review of 0-days Used In-The-Wild in 2019, 2020. URL: https://googleprojectzero.blogspot.com/2020/07/detection-deficit-year-in-review-of-0.html.

[70] Maddie Stone. In-the-Wild Series: Android Post-Exploitation , 2021. URL: https://googleprojectzero.blogspot.com/2021/01/in-wild-series-android-post-exploitation.html.

[71] Maddie Stone. Déjà vu-lnerability – A Year in Review of 0-days Exploited In-The-Wild in 2020, 2021. URL: https://googleprojectzero.blogspot.com/2021/02/deja-vu-lnerability.html.

[72] Maddie Stone. A Very Powerful Clipboard: Analysis of a Samsung in-the-wild exploit chain, 2022. URL: https://googleprojectzero.blogspot.com/2022/11/a-very-powerful-clipboard-samsung-in-the-wild-exploit-chain.html.

[73] Maddie Stone. The More You Know, The More You Know You Don't Know, 2022. URL: https://googleprojectzero.blogspot.com/2022/04/the-more-you-know-more-you-know-you.html.

[74] Maddie Stone. 2022 0-day In-the-Wild Exploitation...so far, 2023. URL: https://googleprojectzero.blogspot.com/2022/06/2022-0-day-in-wild-exploitationso-far.html.

[75] Maddie Stone. The Ups and Downs of 0-days: A Year in Review of 0-days Exploited In-the-Wild in 2022, 2023. URL: https://security.googleblog.com/2023/07/the-ups-and-downs-of-0-days-year-in.html.

[76] Maddie Stone, Jared Semrau, and James Sadowski. We're All in this Together: A Year in Review of Zero-Days Exploited In-the-Wild in 2023, 2024. URL: https://storage.googleapis.com/gweb-uniblog-publish-prod/documents/Year_in_Review_of_ZeroDays.pdf.

[77] Seyed Mohammadjavad Seyed Talebi, Zhihao Yao, Ardalan Amiri Sani, Zhiyun Qian, and Daniel Austin. Undo workarounds for kernel bugs. In *USENIX Security*, 2021.

[78] Zi Fan Tan, Gulshan Singh, and Eugene Rodionov. Attacking Android Binder: Analysis and Exploitation of CVE-2023-20938, 2024. URL: https://androidoffsec.withgoogle.com/posts/attacking-android-binder-analysis-and-exploitation-of-cve-2023-20938/#unlink-primitive.

[79] Yong Wang. Ret2page: The Art of Exploiting Use-Afer-Free Vulnerabilities in the Dedicated Cache, 2022. URL: https://i.blackhat.com/USA-22/Thursday/US-22-WANG-Ret2page-The-Art-of-Exploiting-Use-After-Free-Vulnerabilities-in-the-Dedicated-Cache.pdf.

[80] Zicheng Wang, Yueqi Chen, and Qingkai Zeng. PET: Prevent Discovered Errors from Being Triggered in the Linux Kernel. In *USENIX Security*, 2023.

[81] Daoyuan Wu, Debin Gao, Eric K. T. Cheng, Yichen Cao, Jintao Jiang, and Robert H. Deng. Towards understanding android system vulnerabilities: Techniques and insights. In *AsiaCCS*, 2019.

[82] Le Wu, Xuen Li, and Tim Xia. ExplosION: The Hidden Mines in the Android ION Driver, 2022. URL: https://i.blackhat.com/Asia-22/Friday-Materials/AS-22-Wu-ExplosION-The-Hidden-Mines.pdf.

[83] Le Wu and Qi Zhang. Game of Cross Cache: Let's win it in a more effective way!, 2024. URL: https://i.blackhat.com/Asia-24/Presentations/Asia-24-Wu-Game-of-Cross-Cache.pdf.

[84] Nicolas Wu. Dirty Pagetable: A Novel Exploitation Technique To Rule Linux Kernel, 2023. URL: https://yanglingxi1993.github.io/dirty_pagetable/dirty_pagetable.html.

[85] Wei Wu, Yueqi Chen, Xinyu Xing, and Wei Zou. KEPLER: Facilitating Control-flow Hijacking Primitive Evaluation for Linux Kernel Vulnerabilities. In *USENIX Security*, 2019.

[86] Wei Wu, Yueqi Chen, Jun Xu, Xinyu Xing, Xiaorui Gong, and Wei Zou. FUZE: Towards Facilitating Exploit Generation for Kernel Use-After-Free Vulnerabilities. In *USENIX Security*, 2018.

[87] Xuan Xing, Eugene Rodionov, Jon Bottarini, Adam Bacchus, Amit Chaudhary, Lyndon Fawcett, and Joseph Artgole. Google & Arm - Raising The Bar on GPU Security, 2024. URL: https://security.googleblog.com/2024/09/google-arm-raising-bar-on-gpu-security.html.

[88] Wen Xu, Juanru Li, Junliang Shu, Wenbo Yang, Tianyi Xie, Yuanyuan Zhang, and Dawu Gu. From collision to exploitation: Unleashing use-after-free vulnerabilities in linux kernel. In *CCS*, 2015.

[89] Zhengzi Xu, Yulong Zhang, Longri Zheng, Liangzhao Xia, Chenfu Bao, Zhi Wang, and Yang Liu. Automatic Hot Patch Generation for Android Kernels. In *USENIX Security*, 2020.

[90] Kyle Zeng, Yueqi Chen, Haehyun Cho, Xinyu Xing, Adam Doupé, Yan Shoshitaishvili, and Tiffany Bao. Playing for K(H)eaps: Understanding and Improving Linux Kernel Exploit Reliability. In *USENIX Security*, 2022.

[91] Kyle Zeng, Zhenpeng Lin, Kangjie Lu, Xinyu Xing, Ruoyu Wang, Adam Doupé, Yan Shoshitaishvili, and Tiffany Bao. RetSpill: Igniting User-Controlled Data to Burn Away Linux Kernel Protections. In *CCS*, 2023.

[92] Google Project Zero. Qualcomm KGSL: reclaimed / in-reclaim objects can still be mapped into VBOs, 2024. URL: https://project-zero.issues.chromium.org/issues/42451701.

[93] Hang Zhang and Zhiyun Qian. Precise and Accurate Patch Presence Test for Binaries. In *USENIX Security*, 2018.

[94] Ye Zhang, Le Wu, Shupeng Gao, and Zheng Huang. Attacking NPUs of Multiple Platforms, 2023. URL: https://i.blackhat.com/EU-23/Presentations/EU-23-Zhang-Attacking-NPUs-of-Multiple-Platforms.pdf.

[95] Zheng Zhang, Hang Zhang, Zhiyun Qian, and Billy Lau. An investigation of the android kernel patch ecosystem. In *USENIX Security*, 2021.

Table 7: Devices which are susceptible to n-day vulnerabilities with the commit message shown in Listings 6 to 10. The symbol ✓ refers to susceptible, † refers to fixed in October 2024, †† refers to fixed in November 2024, and ☆ refers to not tested.

Vendor	Device	Model	OOB read	UAF access			ID
			vuln1 (see 6)	**vuln2** (see 7)	**vuln3** (see 8)	**vuln4** (see 9)	**vuln5** (see 10)
Samsung	Galaxy S23	SM-S911B	†	†	†	†	†
	Galaxy S23+	SM-S916B	†	†	☆	†	†
	Galaxy S23 Ultra	SM-S918B	†	†	☆	†	†
	Galaxy Z Flip5	SM-F731B	†	†	☆	†	†
	Galaxy Z Fold5	SM-F946B	†	†	☆	†	†
Xiaomi	Mix Fold 3	Babylon	✓	✓	☆	✓	✓
	Redmi K70, Poco F6 Pro	Vermeer	††	✓	☆	††	✓
	13 Ultra	Ishtar	✓	✓	☆	✓	✓
	13 Pro	Nuwa	††	✓	☆	††	✓
	13	Fuxi	††	✓	☆	††	✓
Asus	ROG Phone 7 Ultimate, ROG Phone 7	AI2205	✓	✓	☆	✓	✓

```
commit a6b25a6b8b9d1d6dfe1eb743ee39de21485d66da
Author: Ramesh Nallagopu <quic_rnallago@quicinc.com>
Date:  Fri Jun 28 22:17:36 2024 +0530

    dsp-kernel: Fix to avoid untrusted pointer dereference

    Currently, the compat ioctl call distinguishes itself
    using a global flag. If a user sends a compat ioctl call
    followed by a normal ioctl call, it may result in using a
    user passed address as a kernel address in the
    fastrpcdriver. To address this issue, consider localizing
    the compat flag for the ioctl call.
```

Listing 6: **Vuln1:** Git commit message of the OOB read vulnerability CVE-2024-21455 [32].

```
commit 6dab51a3af6f217c1729452fa963d0d3568058ec
Author: Abhishek Singh <quic_abhishes@quicinc.com>
Date:  Tue Mar 5 17:19:52 2024 +0530

    dsp-kernel: use-after-free (UAF) in global maps

    Currently, remote heap maps get added to the global list
    before the fastrpc_internal_mmap function completes the
    mapping. Meanwhile, the fastrpc_internal_munmap function
    accesses the map, starts unmapping, and frees the map
    before the fastrpc_internal_mmap function completes,
    resulting in a use-after-free (UAF) issue. Add the map to
    the list after the fastrpc_internal_mmap function
    completes the mapping.
```

Listing 7: **Vuln2:** Git commit message of the first UAF vulnerability CVE-2024-33060 [34].

```
commit 2096d42a680640f9fcc02272bf58f9cc7fa74576
Author: ANANDU KRISHNAN E <quic_anane@quicinc.com>
Date:  Wed Aug 14 10:39:55 2024 +0530

    msm: adsprpc: Avoid taking reference for group_info

    Currently, the get_current_groups API accesses group info,
    which increases the usage refcount. If the IOCTL using the
    get_current_groups API is called many times, the usage
    counter overflows. To avoid this, access group info
    without taking a reference. A reference is not required as
    group info is not released during the IOCTL call.
```

Listing 8: **Vuln3:** Git commit message of second UAF vulnerability CVE-2024-38402 [21].

```
commit f98ae73093949e9e12f64f28bd6103b5f941d32e
Author: Santosh <quic_ssakore@quicinc.com>
Date:  Tue Nov 19 10:54:19 2024 +0530

    dsp-kernel: Add attribute and flag checks during map
    creation

    A persistence map is expected to hold refs=2 during its
    creation. However, the Fuzzy test can create a persistence
    map by configuring a mismatch between attributes and flags
    using the KEEP MAP attribute and FD NOMAP flags. This sets
    the map reference count to 1. The user then calls
    fastrpc_internal_munmap_fd to free the map since it doesnt
    check flags, which can cause a use-after-free (UAF) for
    the file map and shared buffer. Add a check to restrict
    DMA handle maps with invalid attributes.
```

Listing 9: **Vuln4:** Git commit message of the third UAF vulnerability CVE-2024-49848 [30].

```
commit 29cbad25d9bf36341131dcc7dfff75b4255d2111
Author: Abhishek Singh <quic_abhishes@quicinc.com>
Date:  Fri Jun 21 16:04:09 2024 +0530

    dsp-kernel: Do not search the global map in the process-
    specific list

    If a user makes the ioctl call for the
    fastrpc_internal_mmap with the global map flag, fd, and va
    corresponding to some map already present in the process-
    specific list, then this map present in the process-
    specific list could be added to the global list. Because
    global maps are also searched in the process-specific
    list. If a map gets removed from the global list and
    another concurrent thread is using the same map for a
    process-specific use case, it could lead to a use-after-
    free. Avoid searching the global map in the process-
    specific list.
```

Listing 10: **Vuln5:** Git commit message of the ID vulnerability CVE-2024-33060 wrongly assigned [31].